For $\phi(1) = 1$ we should recover the second order accurate Lax-Wendroff scheme ($\omega = -1$ in (10.12)) which suggests that the slope δu_j must be chosen as

$$\delta u_j^n = \begin{cases} \Delta^+ \bar{u}_j^n & a > 0 \\ \Delta^- \bar{u}_j^n & a < 0 \end{cases}.$$

Hence, the general form of the slope limiter should be

$$\overline{\delta u_j} = \psi(\Delta^- \bar{u}_j^n, \Delta^+ \bar{u}_j^n) = \begin{cases} \psi(r_j^{-1})\Delta^- \bar{u}_j^n & a > 0 \\ \psi(r_j)\Delta^+ \bar{u}_j^n & a < 0 \end{cases}, \tag{10.14}$$

where $\psi(a,b)$ or $\psi(r)$ plays the role of the slope limiter and

$$r_j = \frac{\Delta^- \bar{u}_j}{\Delta^+ \bar{u}_j}.$$

The minmod-function (10.3) is an example of this. Comparing (10.13) and (10.14), it is clear that there is a direct relation between the two forms of limiting for the linear case. For the nonlinear case, this direct relation no longer exists. ∎

With the insight gained in the previous example, we can define slope limiter functions ψ to ensure TVD-stability along the lines of (10.3).

The simplest minmod slope limiter is given as

$$\psi_{mm}(a,b) = \text{minmod}(a,b),$$

while a slightly generalized version, known as the MUSCL slope limiter, takes the form [49]

$$\psi_{mu}(a,b) = \text{minmod}(\frac{a+b}{2}, 2a, 2b).$$

The Superbee limiter [39, 34] is given as

$$\psi_{sb}(a,b) = \text{minmod}(\text{maxmod}(a,b), \text{minmod}(2a, 2b)),$$

where the maxmod-function is defined analogously to the minmod function. Its implementation is illustrated in maxmod.m.

Script 10.8. *maxmod.m: Implementation of the maxmod function.*

```
function mfunc = maxmod(v)
% function mfunc = maxmod(v)
% Purpose: Implement the maxmod function on vector v
N = size(v,1); m = size(v,2); psi = zeros(N,1);
s = sum(sign(v),2)/m; ids = find(abs(s)==1);
if(~isempty(ids))
  psi(ids) = s(ids).*max(abs(v(ids,:)),[],2);
end
return;
```

Finally, an often used slope limiter is the van Albada limiter on the form [42]

$$\psi_{va}(a,b) = \frac{(a^2+c^2)b + (b^2+c^2)a}{a^2+b^2+2c^2}.$$

and evaluate the minmod-based indicator

$$\overline{\tilde{u}_j^+} = \bar{u}_j - \text{minmod}(\bar{u}_j - u_{j-1/2}^+, \Delta^+\bar{u}_j, \Delta^-\bar{u}_j), \tag{12.80}$$
$$\overline{\tilde{u}_j^-} = \bar{u}_j + \text{minmod}(u_{j+1/2}^- - \bar{u}_j, \Delta^+\bar{u}_j, \Delta^-\bar{u}_j).$$

If $\overline{\tilde{u}_j^\pm}$ is different from $\tilde{u}_j^\pm$, the local solution is reduced to a linear solution and slope limiting applied. On the other hand, if no limiting is needed, the full polynomial solution is used. This approach retains TVD/TVB-stability in non-smooth regions, while achieving full order accuracy in smooth parts of the solution.

The implementation of the TVD slope-limiting for a general piecewise polynomial function is illustrated in SlopeLimitCSDG.m.

Script 12.13. *SlopeLimitCSDG.m: Applycation of TVD slope limiting to a piecewise polynomial solution.*

```
function [ulimit] = SlopeLimitCSDG(x,u,m,h,N,V,iV);
% function ulimit = SlopeLimitCSDG(x,u,m,h,N,V,iV);
% Purpose: Apply slopelimiter by Cockburn-Shu (1989)
% to u - an m'th order polynomial
eps0=1.0e-8;

% Strength of slope limiter - Minmod: theta=1, MUSCL: theta=2
theta=2.0;

% Compute cell averages and cell centers
uh = iV*u; uh(2:(m+1),:)=0; uavg = V*uh; ucell = uavg(1,:);
    ulimit = u;

% Extend cell averages
[ve] = extendDG(ucell,'N',0,'N',0);

% extract end values and cell averages for each element
uel = u(1,:); uer = u(end,:);
vj = ucell; vjm = ve(1:N); vjp = ve(3:N+2);

% Find elements that require limiting
vel = vj - minmod([(vj-uel),(vj-vjm),(vjp-vj)]);
ver = vj + minmod([(uer-vj),(vj-vjm),(vjp-vj)]);
ids = find(abs(vel-uel)>eps0 | abs(ver-uer)>eps0);

% Apply limiting when needed
if(~isempty(ids))
  % create piecewise linear solution for limiting on specified
    elements
  uhl = iV*u(:,ids); uhl(3:(m+1),:)=0; ulin = V*uhl;
  ux = 2/h*(vj(ids)-ulin(1,:));

  % Limit function
  x0h = ones(m+1,1)*(x(end,:)+x(1,:))/2;
  ulimit(:,ids) = ones(m+1,1)*vj(ids)+(x(:,ids)-x0h(:,ids)).*(
    ones(m+1,1)*...
      minmod([ux(1,:),theta*(vjp(ids)-vj(ids))./h,...
```

```
            theta*(vj(ids)-vjm(ids))./h]));
end
return
```

A slightly more aggressive limiter is proposed in [11]. As a first step, the slope is limited as

$$\overline{\delta u_j} = \text{minmod}\left(\delta u_j, \theta\Delta^+ \bar{u}_j, \theta\Delta^- \bar{u}_j\right).$$

Since this is similar to the TVD approach discussed above, it reduces the accuracy to first-order at local extrema. To address this, it is proposed to consider

$$\overline{\overline{\delta u_j}} = \text{minmod}\left(\delta u_j, \delta u_{j-1}, \delta u_{j+1}\right),$$

and modify the definition of the slopes as

$$\overline{\delta u_j} = \text{maxmod}\left(\overline{\delta u_j}, \overline{\overline{\delta u_j}}\right).$$

The maxmod function is intended to eliminate the local loss of resolution without introducing artificial oscillations. While there is no known theoretical justification, qualitative arguments and extensive computational results presented in [11] support the claim of stability.

The implementation of this more aggressive slope limiting for a general piecewise polynomial function is illustrated in SlopeLimitBSBDG.m.

Script 12.14. *SlopeLimitBSBDG.m: Application of TVD-slope limiting to piecewise polynomial solution, following [11].*

```
function [ulimit] = SlopeLimitBSBDG(x,u,m,h,N,V,iV);
% function ulimit = SlopeLimitBSBDG(x,u,m,h,N,V,iV);
% Purpose: Apply slopelimiter by Burbeau-Sagaut-Bruneau (2001)
% to u - an m'th order polynomial
eps0=1.0e-8;

% Strength of slope limiter - Minmod: theta=1, MUSCL: theta=2
theta=2.0;

% Compute cell averages and cell centers
uh = iV*u; uhx = uh; uh(2:(m+1),:)=0; uavg = V*uh; ucell = uavg
    (1,:);
uhx(3:(m+1),:)=0; ulin = V*uhx; ux = 2/h*(ucell - ulin(1,:));
    ulimit = u;

% Extend cell averages
[ve] = extendDG(ucell,'P',0,'P',0); [vxe] = extendDG(ux,'P',0,'P
    ',0);

% extract end values and cell averages for each element
uel = u(1,:); uer = u(end,:); vj = ucell; vjm = ve(1:N); vjp =
    ve(3:N+2);
vxj = ux; vxjm = vxe(1:N); vxjp = vxe(3:N+2);

% Find elements that require limiting
```

```
vel = vj - minmod([(vj-uel);(vj-vjm);(vjp-vj)]);
ver = vj + minmod([(uer-vj);(vj-vjm);(vjp-vj)]);
ids = find(abs(vel-uel)>eps0 | abs(ver-uer)>eps0);

% Apply limiting when needed
if(~isempty(ids))
  % create piecewise linear solution for limiting on specified
    elements
  uhl = iV*u(:,ids); uhl(3:(m+1),:)=0; ulin = V*uhl;
  x0h=ones(m+1,1)*(x(end,:)+x(1,:))/2;

  % Limit function
  ux1 = minmod([vxj(ids);theta*(vjp(ids)-vj(ids))./h;...
    theta*(vj(ids)-vjm(ids))./h]);
  ux2 = minmod([vxj(ids);vxjm(ids);vxjp(ids)]);
  ulimit(:,ids) = ones(m+1,1)*vj(ids)+(x(:,ids)-x0h(:,ids)).*...
      (ones(m+1,1)*maxmod([ux1;ux2]));
end
return
```

An alternative approach to high-order limiting was first proposed in [7], and subsequently further developed and refined in [11]. Let us begin by expressing the local solution as

$$u_j(x) = \sum_{i=0}^{m} \tilde{u}_{j,i} P_i(x),$$

where P_i is the Legendre polynomial and, for sake of simplicity only, we restrict ourselves to $x \in [-1,1]$. It is clear that $\bar{u}_j = \tilde{u}_{j,0}$. To ensure TVD/TVB-stability, the limiting of the slope

$$(u_j)_x(x) = \sum_{i=1}^{m} \tilde{u}_{j,i} P_i(x),$$

involves the choice of $\tilde{u}_{j,i} = 0$. A slightly different approach involves the individual moments, $\tilde{u}_{j,i}$ of the solution. To ensure monotonicity of the momensts, these are limited as

$$\sqrt{(2i+1)(2i+3)}\tilde{u}_{j,i+1} = \text{minmod}\Big(\sqrt{(2i+1)(2i+3)}\tilde{u}_{j,i+1}, \theta(\tilde{u}_{j+1,i} - \tilde{u}_{j,i}), \theta(\tilde{u}_{j,i} - \tilde{u}_{j-1,i})\Big),$$

where $\theta \geq 0$ is a free parameter. This approach is applied in a backward fashion to the moments of decreasing order, i.e., $i = m-1, \ldots, 0$. Furthermore, it is adaptive in the sense that once a moment is not limited, no lower moments are altered.

The implementation of hierarchical moment limiting for a general piecewise polynomial function is illustrated in MomentLimitDG.m.

Script 12.15. *MomentLimitDG.m: Routine for applying moment limiting to piecewise polynomial solution*

```
function [ulimit] = MomentLimitDG(x,u,m,h,N,V,iV);
% function ulimit = MomentLimitDG(x,u,m,h,N,V,iV);
% Purpose: Apply moment limiter to u - an m'th order polynomial
eps0=1.0e-8; eps1 = 1.0e-8;
```

```
% Strength of slope limiter - Minmod: theta=1, MUSCL: theta=2
theta=2.0;

% Compute cell averages and cell centers
uh = iV*u; uh(2:(m+1),:)=0; uavg = V*uh; ucell = uavg(1,:);
    ulimit = u;

% Extend cell averages
[ve] = extendDG(ucell,'N',0,'N',0);

% extract end values and cell averages for each element
uel = u(1,:); uer = u(end,:); vj = ucell; vjm = ve(1:N); vjp =
    ve(3:N+2);

% Find elements that require limiting
vel = vj - minmod([(vj-uel)';(vj-vjm)';(vjp-vj)']);
ver = vj + minmod([(uer-vj)';(vj-vjm)';(vjp-vj)']);
ids = (abs(vel-uel)<eps1 & abs(ver-uer)<eps1);
mark = zeros(1,N); mark = (ids | mark);

% Compute expansion coefficients
uh = iV*u;

% Apply limiting when needed
for i=m+1:-1:2
  uh1 = uh(i,:); uh2 = uh(i-1,:);
  [uh2e] = extendDG(uh2,'P',0,'P',0); uh2m = uh2e(1:N); uh2p =
    uh2e(3:N+2);
  con = sqrt((2*i+1)*(2*i-1));
  uh1 = 1/con*minmod([con*uh1'; theta*(uh2p - uh2)'; ...
          theta*(uh2 - uh2m)'])'.*(1-mark) + mark.*uh1';
  idsh = abs(uh1-uh(i,:))<eps0; mark = (idsh | mark);
  uh(i,:) = uh1;
end
ulimit = V*uh;
return
```

The scheme combines simplicity with the use of all available high-order information. However, no stability theory is known.

This same approach can be applied to the more aggressive limiter, and extended to higher order by introducing the limiting function

$$\sqrt{(2i+1)(2i+3)}\tilde{u}_{j,i+1} = \text{minmod}\Big(\sqrt{(2i+1)(2i+3)}\tilde{u}_{j,i+1}, \theta(w^{+}_{j+1/2} - \tilde{u}_{j,i}), \theta(\tilde{u}_{j,i} - w^{-}_{j-1/2})\Big),$$

where

$$w^{\pm}_{j\pm1/2} = \tilde{u}_{j\pm1,i} \mp \sqrt{(2i+1)(2i+3)}\tilde{u}_{j\pm1,i+1}.$$

In the case of systems, these techniques should generally be applied to the characteristic variables in order to avoid oscillations, as discussed in Chapter 11.3.4. However, for some problems it may suffice to apply limiting to the conserved variables [7, 11].

```
% Set problem parameters
xmin = -1.0; xmax = 1.0;
FinalTime = sqrt(2.0); CFL = 0.25;
epl = 1.0; mul = 1.0; epr = 2.0; mur = 1.0;

% Generate mesh
VX = (xmax-xmin)*(0:N)/N + xmin; r = LegendreGL(m);
x = ones(m+1,1)*VX(1:N) + 0.5*(r+1)*(VX(2:N+1)-VX(1:N));
h = (xmax-xmin)/N;

% Define domain, materials and initial conditions
Ef = zeros(m+1,N); Hf = zeros(m+1,N);
for k = 1:N
   [Ef(:,k), Hf(:,k), ep(:,k), mu(:,k)] = ...
       CavityExact(x(:,k), epl, epr, mul, mur, 0);
end

% Set up material parameters
eps1 = [epl*ones(1,N/2), epr*ones(1,N/2)];
mu1 = [mul*ones(1,N/2), mur*ones(1,N/2)];
ep = ones(m+1,1)*eps1; mu = ones(m+1,1)*mu1;

% Solve Problem
q = [Ef Hf]; q = zeros(m+1,N,2); q(:,:,1) = Ef; q(:,:,2) = Hf;
[q] = MaxwellDG1D(x,q,ep,mu,h,m,N,CFL,FinalTime);
```

In MaxwellDG1D.m, the system is integrated in time using a third order SSP-RK scheme. This relies on the evaluation of the right hand side, discretized using the discontinuous Galerkin method, as illustrated in MaxwellDGrhs1D.m.

Script 12.27. *MaxwellDG1D.m: Time-integration routine for Maxwell equations using the discontinuous Galerkin method.*

```
function [q] = MaxwellDG1D(x,q,ep,mu,h,m,N,CFL,FinalTime)
% function [q] = MaxwellDG1D(x,q,ep,mu,h,m,N,CFL,FinalTime)
% Purpose  : Integrate 1D Maxwells equation until FinalTime
    using a DG
%            scheme and a 3rd order SSP-RK method.
% Initialize operators at Legendre Gauss Lobatto grid
r = LegendreGL(m); V  = VandermondeDG(m, r); D = DmatrixDG(m, r,
    V);
Ma = inv(V*V'); S = Ma*D;

% Initialize extraction vector
VtoE = zeros(2,N);
for j=1:N
  VtoE(1,j) = (j-1)*(m+1)+1; VtoE(2,j) = j*(m+1);
end

% Compute smallest spatial scale timestep
rLGLmin = min(abs(r(1)-r(2)));
time = 0; tstep = 0;

% Set timestep
cvel = 1./sqrt(ep.*mu); maxvel = max(max(cvel));
```

```
k = CFL*rLGLmin*h/2/maxvel;

% integrate scheme
while (time<FinalTime)
  if (time+k>FinalTime) k = FinalTime-time; end

  % Update solution
  [rhsq]  = MaxwellDGrhs1D(x,q,ep,mu,h,k,m,N,Ma,S,VtoE,maxvel);
  q1 = q + k*rhsq;
  [rhsq]  = MaxwellDGrhs1D(x,q1,ep,mu,h,k,m,N,Ma,S,VtoE,maxvel);
  q2 = (3*q + q1 + k*rhsq)/4;
  [rhsq]  = MaxwellDGrhs1D(x,q2,ep,mu,h,k,m,N,Ma,S,VtoE,maxvel);
  q  = (q + 2*q2 + 2*k*rhsq)/3;
  time = time+k; tstep = tstep+1;
end
return
```

Script 12.28. *MaxwellDGrhs1D.m: Evaluation of the right-hand-side for solving the one-dimensional Maxwell equations using a discontinuous Galerkin method.*

```
function [rhsq] = MaxwellDGrhs1D(x,q,ep,mu,h,k,m,N,Ma,S,VtoE,
    maxvel);
% function [dq] = MaxwellDGrhs1D(x,q,ep,mu,h,k,m,Ma,Sr,VtoE,
    maxvel);
% Purpose: Evaluate right hand side for Maxwells equation using
    DG method
Imat = eye(m+1); Ee = zeros(2,N+2); He = zeros(2,N+2);
EMl = zeros(N,2); EMr = zeros(N,2); EMm = zeros(N,2); EMp =
    zeros(N,2); E = q(:,:,1); H = q(:,:,2);

% Impose boundary conditions
[Ee] = extendDG(E(VtoE,1),'D',0,'D',0);
[He] = extendDG(H(VtoE,2),'N',0,'N',0);

% Compute numerical fluxes at interfaces
EMr = [Ee(2,2:N+1)' He(2,2:N+1)']; EMl = [Ee(1,2:N+1)' He(1,2:N
    +1)'];
EMm = [Ee(2,1:N)' He(2,1:N)']; EMp = [Ee(1,3:N+2)' He(1,3:N+2)
    '];
fluxr = MaxwellLF(EMr,EMp,ep(1,:)',mu(1,:)',k/h,maxvel)';
fluxl = MaxwellLF(EMm,EMl,ep(1,:)',mu(1,:)',k/h,maxvel)';

% Compute right hand side of Maxwell's equation
rE = S'*H./ep - (Imat(:,m+1)*fluxr(1,:) - Imat(:,1)*fluxl(1,:));
rH = S'*E./mu - (Imat(:,m+1)*fluxr(2,:) - Imat(:,1)*fluxl(2,:));
rhsq = (h/2*Ma)\[rE rH]; rhsq(:,:,1) = (h/2*Ma)\rE; rhsq(:,:,2) = (h/2*Ma)\rH;
return
```

It is clear that, if we design the grid such that the material interface is within an element, the limited regularity of the solution will impact the overall achievable accuracy. Based on the theoretical developments in Chap. 12.1.1, we expect $\mathcal{O}(h^{3/2})$ in such a case, in agreement with the computational results in Chapter 5.3.4.